15-122: Principles of Imperative Computation, Spring 2013

Homework 4 Programming: Editorlab

Due: Thursday, March 7, 2013 by 23:59

For the programming portion of this week's homework, you'll implement the core data structure for a text editor: the gap buffer. You will write three C0 files corresponding to different tasks:

- `gap-buffer.c0` (described in Section 2)
- `text-buffer.c0` (described in Section 3)
- `text-editor.c0` (described in Section 4)

Be sure to *read the specifications carefully*—you'll thank yourself later!

You should submit these files electronically by 11:59 pm on the due date. Detailed submission instructions can be found below.

Assignment: Text Editor (30 points)

Starter code. Download the file `hw4-handout.tgz` from the course website. When you untar it, you will find the following files:

`gap-buffer.c0`	Gap buffer data structure
`text-buffer.c0`	Doubly Linked List of gap buffers
`text-editor.c0`	Text editing functionality

Additional files are provided for testing and for running the E0 editor; these will be discussed when they come up.

Compiling and running. You will compile and run your code using the standard C0 tools. Once you have completed all the parts of the assignment, William Lovas's E0 text editor can be compiled and run as follows:

```
% cc0 -d -o E0 lovas-E0.c0
% ./E0
```

Submitting. Once you've completed some files, you can submit them to Autolab. There are two ways to do this:

From the terminal on Andrew Linux (via cluster or ssh) type:

```
% handin hw4 gap-buffer.c0 text-buffer.c0 text-editor.c0
```

Your score will then be available on the Autolab website.

Your files can also be submitted to the web interface of Autolab. To do so, please `tar` them, for example:

```
% tar -czvf sol.tgz gap-buffer.c0 text-buffer.c0 text-editor.c0
```

Then, go to `https://autolab.cs.cmu.edu/15122-s13` and submit them as your solution to homework 4 (editorlab).

You may submit your assignment up to 25 times. When we grade your assignment, we will consider the most recent version submitted before the due date. If you get any errors while trying to submit your code, you should contact the course staff immediately.

Annotations. Be sure to include appropriate `//@requires`, `//@ensures`, `//@assert`, and `//@loop_invariant` annotations in your program. You should provide loop invariants and any assertions that you use to check your reasoning. If you write any auxiliary functions, include precise and appropriate pre- and postconditions.

You should write these as you are writing the code rather than after you're done: documenting your code as you go along will help you reason about what it should be doing, and thus help you write code that is both clearer and more correct. **Annotations are part of your score for the programming problems; you will not receive maximum credit if your annotations are weak or missing.**

Style. Strive to write code with *good style*: indent every line of a block to the same level, use descriptive variable names, keep lines to 80 characters or fewer, document your code with comments, etc. If you find yourself writing the same code over and over, you should write a separate function to handle that computation and call it whenever you need it. We will read your code when we grade it, and good style is sure to earn our good graces. Feel free to ask on Piazza if you're unsure of what constitutes good style.

Task 0 (5 pts) *5 points on this assignment will be given for style.*

1 Overview: A text editor based on gap buffers

In this assignment you will implement a simple text editor based on the *gap buffer* technique. A gap buffer is a generalization of an unbounded array: although an unbounded array allows for efficient insertion and deletion of elements from the end, a gap buffer allows for efficient insertion and deletion of elements from the middle.

Given an array that is only half filled, adding items after the last item currently in the array requires very little work, because this simply means placing it in the next unused index and increasing the `size`. However, there is no unused space in the middle. The only way to place a new item in the middle is to shift elements over to make room for the new item.

A gap buffer attempts to overcome the overhead of shifting by placing the empty portion of the array in the middle. Hence the name “gap buffer" referring to the “gap" in the middle of the “buffer". The gap is not fixed to any one position. At any time it could be in the middle of the buffer or just at the beginning or anywhere in the buffer. We can immediately see the potential benefits of this approach.

By moving the **cursor** in the text editor, we are automatically moving a gap, and thereby providing the unused portion of the array to be used for possible insertions. In the worst case scenario we have to move the gap from the beginning of the text file to the end. But if subsequent operations are only a few indexes apart, we will get a lot better performance compared to using a dynamic array. This is why it is said that the gap buffer technique increases performance of repetitive operations occurring at relatively close indexes. We claim without proof that the amortized cost of insertion into the gap buffer is constant.

Implementing a text editor as just one gap buffer is not particularly realistic. One large edit buffer requires the entire file contents to be stored in a single, contiguous block of memory, which can be difficult to allocate for large files. Instead, a more realistic strategy is to combine the gap buffer technique with a doubly linked list. The benefit of a linked list is that it allows the file to be split across several chunks of memory. Therefore, in this assignment we will represent a text editor as a doubly linked list where each node contains a fixed-size (16 characters) gap buffer. The contents of a text file represented in this way is simply the concatenation of the contents of each gap buffer in the linked list.

Warning: While the dictionaries you implemented in Claclab were tested as an opaque interface (black-box testing), the data structures you implement in this assignment will be tested as a white-box implementation. The editor does not respect the interface perfectly, and so you must implement the data structures as this assignment describes.

2 Gap Buffer

A gap buffer is an array of characters. The array is logically split into two segments of text - one at the beginning of the array, which grows upwards, and one at the end which grows downwards. The space between those two segments is called the *gap*. The gap is the **cursor** in the buffer. To move the cursor forwards, you just move the gap (assuming that the gap is not empty). To insert a character after the cursor, you place it at the beginning of the gap. To delete the character before the cursor, you just expand the gap.

To implement a gap buffer there are a couple bits of information that we need to keep track of. A gap buffer is represented in memory by an array of elements stored along with its size (`limit`) and two integers representing the beginning (inclusive, `gap_start`) and end (exclusive, `gap_end`) of the gap (see Figure 1).

```
typedef struct gap_buffer* gapbuf;
struct gap_buffer {
    int limit;          /* limit > 0                     */
    char[] buffer;      /* \length(buffer) == limit      */
    int gap_start;      /* 0 <= gap_start                */
    int gap_end;        /* gap_start <= gap_end <= limit */
};
```

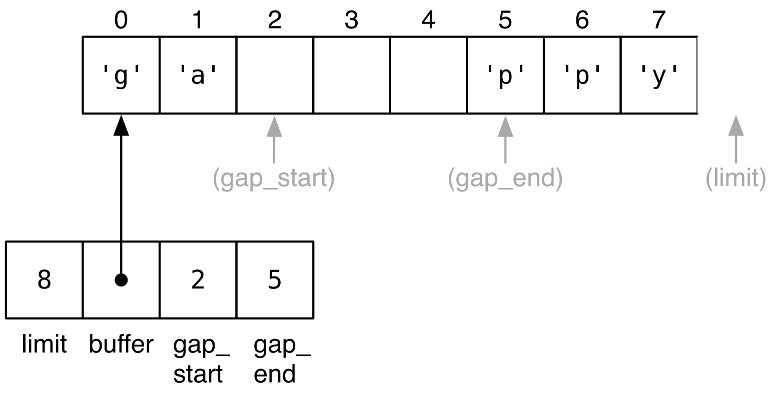

Figure 1: A gap buffer in memory.

Task 1 (2 pts) *A valid gap buffer is non-NULL, has a strictly positive limit which correctly describes the size of its array, and has a gap start and gap end which are valid for the array. Implement the function*

```
bool is_gapbuf(gapbuf G)
```

that formalizes the gap buffer data structure invariants.

Text buffers may allow a variety of editing operations to be performed on them; for the purposes of this assignment, we'll consider only four operations: move forward a character, move backward a character, insert a character, and delete a character. As an example, below is a diagram of a gap buffer which is an array of characters with a gap in the middle (situated between the "p" and the "a" in "space"):

```
the sp[..]ace race
```

To move the gap (the cursor in the text editor) forward, we copy a character across it:

```
the spa[..]ce race
```

To delete a character (before the cursor), we simply expand the gap:

```
the sp[...]ce race
```

To insert a character (say, "i"), we write it into the left side of the gap (shrinking it by one):

```
the spi[..]ce race
```

The gap can be at the left end of the buffer,

```
[..]the space race
```

or at the right end of the buffer,

```
the space race[..]
```

and a buffer can be empty,

```
[................]
```

or it can be full (this depends on the buffer size (limit))

```
the space ra[]ce!!
```

Note that in an emacs-like interface, where the cursor appears as a highlighted character in the buffer, the cursor will display on the character immediately following the gap. So following the examples above,

```
the sp[..]ace race
```

would display as:

```
the space race
```

while

```
the space race[..]
```

would display as:

```
the space race█
```

In the above illustrations, we use dots to indicate spots in the gap buffer whose contents we don't care about. Those spots in the gap buffer don't need to contain the default character `'\0'` or the character `'.'` or anything else in particular.

Task 2 (4 pts) *Implement the following utility functions on gap buffers:*

Function:	***Returns true iff...***
`bool gapbuf_empty(gapbuf G)`	*. . . the gap buffer is empty*
`bool gapbuf_full(gapbuf G)`	*. . . the gap buffer is full*
`bool gapbuf_at_left(gapbuf G)`	*. . . the gap is at the left end of the buffer*
`bool gapbuf_at_right(gapbuf G)`	*. . . the gap is at the right end of the buffer*

Task 3 (4 pts) *Implement the following interface functions for manipulating gap buffers:*

`gapbuf gapbuf_new(int limit)`	*Create a new gapbuf of size limit*
`void gapbuf_forward(gapbuf G)`	*Move the gap forward, to the right*
`void gapbuf_backward(gapbuf G)`	*Move the gap backward, to the left*
`void gapbuf_insert(gapbuf G, char c)`	*Insert the character c before the gap*
`void gapbuf_delete(gapbuf G)`	*Delete the character before the gap*

See page 6 for details. If an operation cannot be performed (e.g., moving the gap backward when it's already at the left end), a contract should fail.

All functions should require and ensure the data structure invariants. Furthermore, the gap buffer returned by `gapbuf_new` should be empty. Use these facts to help you write your code, and document them with appropriate assertions.

2.1 Testing

You can test your gap buffer implementation interactively by compiling and running the provided `gap-buffer-test.c0`; you are encouraged to use this file as a starting point for writing your own unit tests.

```
% cc0 -d -o gap-buffer-test gap-buffer-test.c0
% ./gap-buffer-test
```

Try entering "`space race<<<<<<<<^p<<the >>^p>>>>>>>>!!<<<<<<<^^^^^great`" and seeing what happens.

3 Doubly-Linked Lists

Another data structure that will be used to represent an edit buffer is a *doubly-linked list*. We have seen singly-linked lists used to represent stacks and queues—sequences of nodes, each node containing some data and a pointer to the next node. The nodes of a doubly-linked list contain a `data` field just like those of a singly-linked list, but in contrast, the doubly-linked nodes contain *two* pointers: one to the next element (`next`) and one to the *previous* (`prev`).

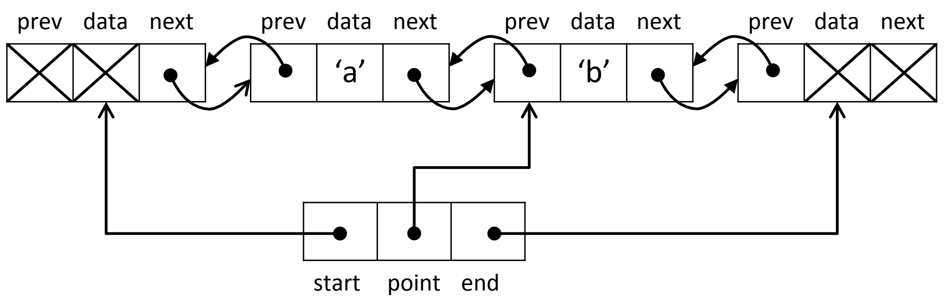

Figure 2: An editable sequence as a doubly-linked list in memory.

An editable sequence is represented in memory by a doubly-linked list and three pointers: one to the `start` of the sequence, one to the `end` of the sequence, and one to the distinguished `point` node where updates may take place (see Figure 2). We employ our usual trick of terminating the list with "dummy" nodes whose contents we never inspect.

```
typedef struct list_node* dll;
struct list_node {
    elem data;
    dll next;
    dll prev;
};

typedef struct text_buffer* tbuf;
struct text_buffer {
    dll start;
    dll point;
    dll end;
};
```

We can visualize a doubly-linked list as the sequence of its `data` elements with terminator nodes at both ends and one distinguished element, called `point`:

$$\texttt{START <--> 'a' <-->} \boxed{\texttt{'b'}} \texttt{ <--> END}$$

For now, we do not concern ourselves with the type of the `data` elements: basic doubly-linked list functions are agnostic to it anyway. (The picture above treats the data elements as C0 characters.)

Task 4 (4 pts) *A valid doubly-linked list has the following properties:*

- *the* `next` *links proceed from the* `start` *node to the* `end` *node, passing* `point` *node along the way*
- *the* `prev` *links mirror the* `next` *links*
- `point` *is distinct from both the* `start` *and the* `end` *nodes, i.e., the list is non-empty*

Implement the function

```
bool is_linked(tbuf B)
```

that formalizes the linking invariants on a doubly-linked list text buffer. You are not required to check for circularity, but you may find it to be a useful exercise.

This task is not trivial. There are many ways for a doubly-linked list to be invalid, even without circularity. For instance, your `is_linked` function will be tested against structures with `NULL` pointers in various locations and against almost-correct doubly-linked lists like the ones in Figure 3 and Figure 4. Your code should return `false` in each case.

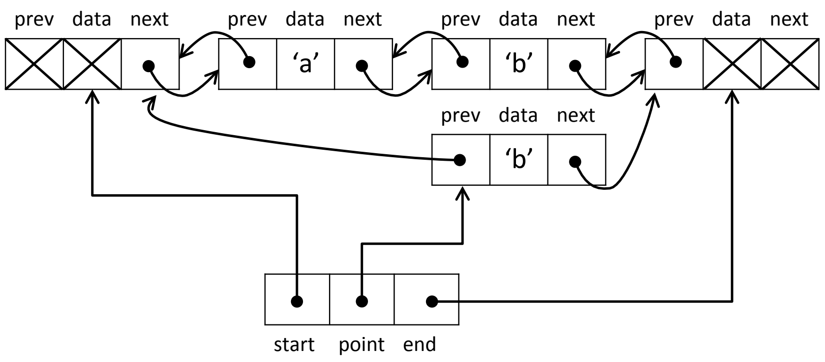

Figure 3: Not a doubly-linked list (the `point` isn't on the path from `start` to `end`).

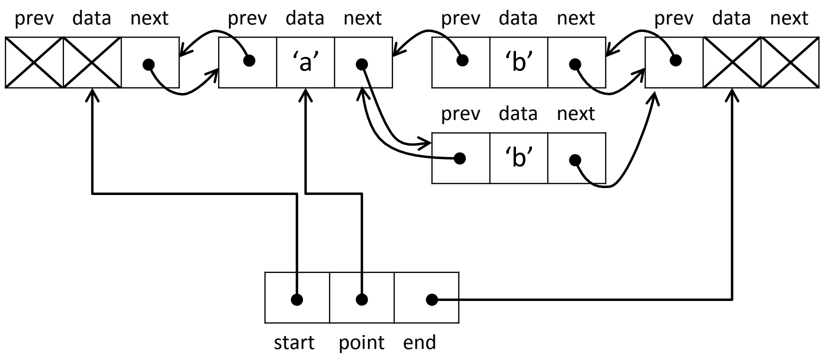

Figure 4: Not a doubly-linked list (the `prev` links don't mirror the `next` links).

Task 5 (3 pts) *Implement the following utility functions on doubly-linked text buffers:*

Function:	***Returns true iff...***
`bool tbuf_at_left(tbuf B)`	*. . . the point is at the far left end*
`bool tbuf_at_right(tbuf B)`	*. . . the point is at the far right end*

and the following interface functions for manipulating doubly-linked text buffers:

`void tbuf_forward(tbuf B)`	*Move the point forward, to the right*
`void tbuf_backward(tbuf B)`	*Move the point backward, to the left*
`void tbuf_delete_point(tbuf B)`	*Remove the point node from the list*

As above, if an operation cannot be performed, a contract should fail. When deleting the point, the new point may be either to the right or to the left of the old one.

These functions should require and preserve the linking invariant you wrote above, and you should both document this fact and use it to help write the code. Be especially careful when implementing deletion! Note, we cannot delete the point if it is the only non-terminator node.

3.1 Testing

You can test your doubly-linked-list implementation interactively by compiling and running the provided `text-buffer-test.c0`, which treats elements as C0 characters as in the illustration above. You are encouraged to use this file as a starting point for writing your own unit tests.

```
% cc0 -d -o text-buffer-test text-buffer-test.c0
% ./text-buffer-test
```

Try entering “`steady`” as the input word and then “`^<<<<^>>^`” as the series of actions and seeing what happens.

If you write your own test code, make sure that you put either `gap-buffer.c0` (which declares the type `elem` to be `gapbuf`) or `elem-char.c0` (which declares the type `elem` to be `char`) on the command line before `text-buffer.c0`. If you try to compile your `text-buffer.c0` file without first defining what `elem` is, you will probably get an error `"expected a type name, found identifier 'elem'"`, because the C0 compiler assumes `"elem"` is an identifier unless you have already used a `typedef` to explain to C0 that it is really a type name.

4 Putting It Together

Now we will implement a text editor as a doubly-linked list of fixed-size gap buffers (each buffer is 16 characters long). The contents of a text buffer represented in this way is simply the concatenation of the contents of its requisite gap buffers, in order from the `start` to the `end`. The gap representing the text editor's cursor is be the gap at the linked list's `point`. To move the cursor, we use a combination of gap buffer motion and doubly-linked list motion:

`** <--> just_a_[.] <-->` $\boxed{\texttt{j[....]ump}}$ `<--> **`

move ←: `** <--> just_a_[.] <-->` $\boxed{\texttt{[....]jump}}$ `<--> **`

move ←: `** <-->` $\boxed{\texttt{just\_a[.]\_}}$ `<--> [....]jump <--> **`

4.1 Text buffer invariants

There are a lot of invariants that we want to check in this representation. Two very simple ones are that our linked list of gap buffers should be a linked list (`is_linked`), and each element in the linked list should be a gap buffer (`is_gapbuf`).

Another invariant that arises from this representation is that a text buffer must either be the empty text buffer:

`START <-->` $\boxed{\texttt{[................]}}$ `<--> END`

or else all the gap buffers must be non-empty. Additionally, all the gap buffers are themselves well-formed, and they all have the same size, and the size is 16. (We used gap buffers of size 8 for simplicity in some of the examples, but you must use 16 in your implementation.)

Another important invariant is *alignment*. It is easiest to observe on larger cases:

`** <--> well_i[..] <--> sn't_[...] <-->` $\boxed{\texttt{this\_[..]l}}$ `<--> [.....]ong <--> **`

Notice that for all gap buffers to the left of the point, the gap is on the right. Similarly, for all gap buffers to the right of the point, the gap is on the left. We call this invariant alignment. If alignment fails to hold, we will have a very hard time moving the point between gap buffers. If we had this text buffer instead:

`** <--> well_i[..] <--> sn't_[...] <-->` $\boxed{\texttt{this\_[..]l}}$ `<--> ong[.....] <--> **`

then, as we move to the right,

`** <--> well_i[..] <--> sn't_[...] <-->` $\boxed{\texttt{this\_l[..]}}$ `<--> ong[.....] <--> **`

`** <--> well_i[..] <--> sn't_[...] <--> this_l[..] <-->` $\boxed{\texttt{ong[.....]}}$ `<--> **`

we find that the cursor suddenly jumps to the end of the buffer, skipping over "ong" entirely.

Task 6 (2 pts) *A valid text buffer satisfies all the invariants described above: it is a valid doubly-linked list containing valid size-16 gap buffers, it either consists of one empty gap buffer or some number of non-empty gap buffers, and it is aligned. Implement the function*

```
bool is_tbuf(tbuf B)
```

that formalizes the text buffer data structure invariants.

Hint: you may find it easier to decompose `is_tbuf` into multiple functions (such as one that checks alignment and one that checks that the one-empty-or-all-nonempty property).

4.2 Manipulating text buffers

Task 7 (2 pts) *Implement the following utility functions on text buffers:*

Function:	***Returns true iff...***
`bool tbuf_empty(tbuf B)`	*the text buffer is empty*

and a text buffer constructor:

`tbuf tbuf_new()`	*Construct a new, empty text buffer with a gap-buffer-size of 16*

Recall that in order to be aligned, a text buffer must first be a valid text buffer according to `is_tbuf`, all gap buffers to the left ("before") the point must have their gaps on the right, and all gap buffers to the right ("after") the point must have their gaps on the left. Alignment specifies nothing about the properties of the point itself.

To insert into the buffer (of the point node), we have to check if the buffer is full or not. When a gap buffer is full, we split the point node into two nodes. The data in the buffer will be split as well:

```
                 ** <--> splitend[] <--> **

insert 's':  ** <--> spli[....] <--> tends[...] <--> **
```

To split a full gap buffer, we have to copy each half of the character data into one of two new gap buffers, taking special note of where the new gaps should end up. The following diagrams may help you visualize the intended result:

```
full buffer:  abc[]defghABCDEFGH          full buffer:  stuvwxyzSTUV[]WXYZ

splits into:  abc[........]defgh          splits into:  stuvwxyz[........]
              [........]ABCDEFGH                        STUV[........]WXYZ
```

We can then link the new gap buffers into the doubly-linked list, taking care to preserve the text buffer invariants.

Task 8 (2 pts) *Implement a function* `split_point(tbuf B)` *which takes a valid text buffer whose point is full and turns it into a valid text buffer whose point is not full.*

To delete from the buffer we use the gap buffer's `gapbuf_delete` function and when one the buffer becomes empty, we delete it:

```
                START <--> deletio[.] <--> n[........] <--> END

delete:  START <--> deletio[.] <--> END
```

Task 9 (2 pts) *Implement the following interface functions for manipulating text buffers:*

`void forward_char(tbuf B)`	*Move the cursor forward, to the right*
`void backward_char(tbuf B)`	*Move the cursor backward, to the left*
`void insert_char(tbuf B, char c)`	*Insert the character c before the cursor*
`void delete_char(tbuf B)`	*Delete the character before the cursor*

These functions directly respond to a user's input. *That means that if an operation cannot be performed (e.g., pressing the "left" key to move the cursor backward with* `backward_char` *when it's already at the left end), the function should leave the text buffer* **unchanged** *instead of raising an error or assertion violation.*

4.3 Testing

You can test your implementation by compiling and running the provided `text-editor-test.c0` test driver which prints a visual representation of the internal data of a text buffer. You are encouraged to use this file as a starting point for writing your own unit tests.

```
% cc0 -d -o te-test text-editor-test.c0
% ./te-test
     START <--> _[................]_ <--> END
'a': START <--> _a[...............]_ <--> END
'b': START <--> _ab[..............]_ <--> END
...
```

The expected output is stored in `expected.txt`.

Using this driver you can test either your complete implementation or each function independently. The testing is based on printing functionality implemented in `visuals.c0`.

After you've completed your text buffer implementation and tested it thoroughly, you can try it out interactively by compiling against `lovas-E0.c0` – a minimalist text editor front-end written by William Lovas.

Enjoy the hard-won fruits of your careful programming labors!

5 Bonus: Extending the Editor

Extend the E0 editor implementation with some interesting features. A few suggestions to pique your imagination: a better display algorithm, a better splitting algorithm, line motion, more editing commands, copy and paste – be creative! Feel free to extend the data structures in any way necessary to support your changes effectively. Submit your modified implementation as files named `bonus-*.c0` and include a `bonus-README.txt` file explaining your work and how it can be compiled. The entries will be judged both in terms of the interactive editing experience, the supporting data structures and algorithms, and the quality of the code.